Classical and Deep RL on MiniHack

Introduction

In this project I use the **gymnasium** api and the **MiniHack** framework, in order to interact with environments for RL. I implement my own traditional RL algorithms from scratch (**Monte Carlo, SARSA, Tabular Q Learning, Dyna-Q**), and I also use the **stable_baselines3** to work with Deep Reinforcement Learning techniques (DQN and PPO).

The environments I work on are MiniHack environments, that follow specific rules (described on section "MiniHack Familiarisation"). In *minihack_env.py* we have a wrapper function, a function that helps us initialise environments and some des-files we use to create the custom environments.

In *commons.py* we have the basic definitions of 2 Abstract Classes, AbstractAgent and AbstractRLTask, we later on inherit to write more specific classes and implement our algorithms. We also have 2 functions that help us process the observations we get as outputs from the MiniHack environments (mainly for visualisation/rendering). The file named *quickstart.py* demonstrates the use of these functions.

GYM Familiarisation

In this section I familiarise myself with Gymnasium by creating a custom Gymnasium environment class (a simple rectangular grid where the agent is in one corner and the goal on the opposite, every action rewards -1) named Env11; I don't work with MiniHack yet. I also create the RandomAgent which implements what the name implies and the RLTask, both by inheriting the aforementioned abstract classes. The RLTask helps us visualise the process of one episode and the evolution of the RL algorithm. To render one episode, I just print the map tile representations in matrix format for every step (figure 2).

The way we choose to depict the evolution throughout many episodes (figure 1) is by plotting the sequence of the average of the episode returns, $\hat{G}_k = \frac{1}{k+1}\sum_{i=0}^{k} G_i$, k the episode index. By using average returns, we avoid spikes in the curves due to the stochastic nature of the exploration. I depict figures for a $5x5$ grid.

I observe that after a while, the average return seems to converge to a value around -106, these are the steps the random agent needs on average to find the goal on this grid size. The simplistic rendering is able in depicting an agent that takes actions at random. After playing around with these basic implementations, I continue by starting to work with the MiniHack environments that I will later use for RL.

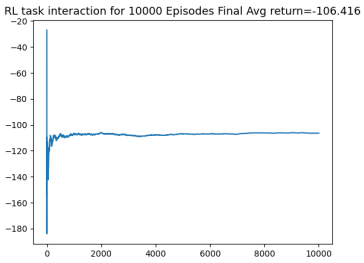


Figure 1: Average Return Plot, 10000 Episodes

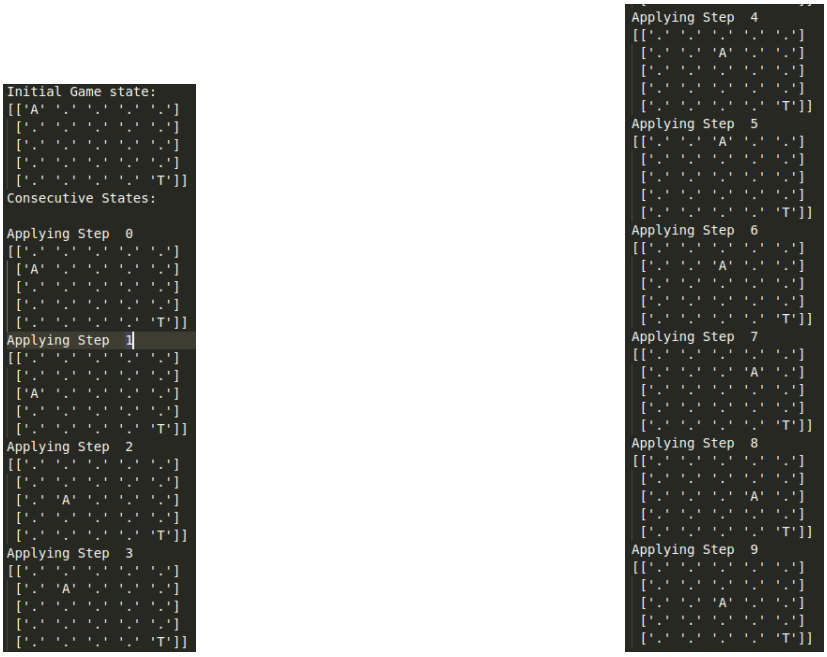
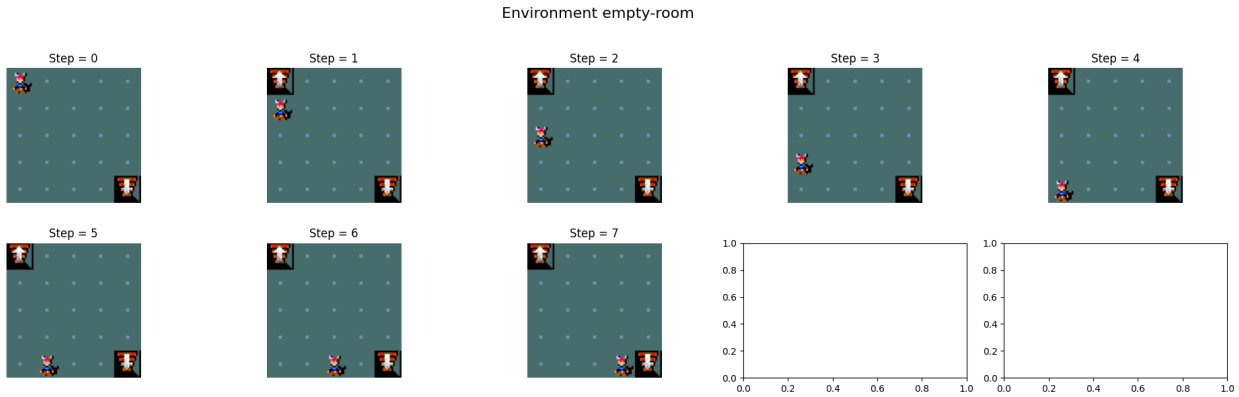


Figure 2: Visualising (Rendering) One Episode for 10 Steps

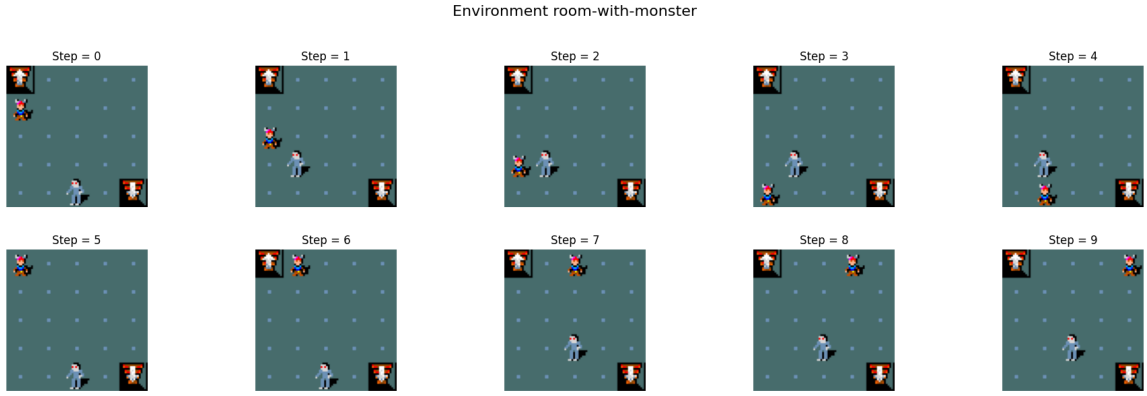
Minihack Familiarisation

In this part, I implement the FixedAgent, which starts by always going down until it cannot anymore (by checking whether the next tile is wall, blank tile, trees or lava), and then always goes right. I then experiment with this agent on the 4 different environments I will use in the project. In these environments, each action rewards -1, reaching the goal (downward stairs, unique in every map) rewards 0 and dying rewards -100 and teleports the agent to the initial position without terminating the episode. The player can die either by stepping on lava or by a monster attack.

In EMPTY_ROOM we have a square map, with the agent and the ladder. In CLIFF we have the cliff environment as described in the Sutton and Barto book. ROOM_WITH_LAVA is an environment that has lava and is more complicated than CLIFF. Finally, ROOM_WITH_MONSTER is a square map with the ladder, the agent and a moving adversary monster.



(a) The agent solves the empty room environment and the episode terminates



(b) The agent dies, is teleported in the initial position and continues to go right

Figure 3: Demonstration of FixedAgent in the room, both empty and with monster.

We can observe the environments on figures (3, 4). The figures showcase that both the environments and the fixed agent function with the way I describe above. Some things that should be noted about the environments are that:

1) We can use the environments both with a Random and non random layout option, which changes the positions of the lava, ladder, player etc. inside the map. The Cliff is demonstrated with random=True here, the others with random=False.

2) With the way the **custom environments** (ROOM_WITH_LAVA, ROOM_WITH_MONSTER, CLIFF - through des) are implemented there are many cases where **the episodes abort** (not truncate !) early, seemingly for no reason. For example, the ROOM_WITH_LAVA nearly always aborts at step 100 in each episode. Also, **for Random=True** in the custom maps there is **no downwards ladder**, so **no way to terminate and reach the agent goal!** (e.g. CLIFF on 4) We should be careful of these when designing and studying the experiments later on.

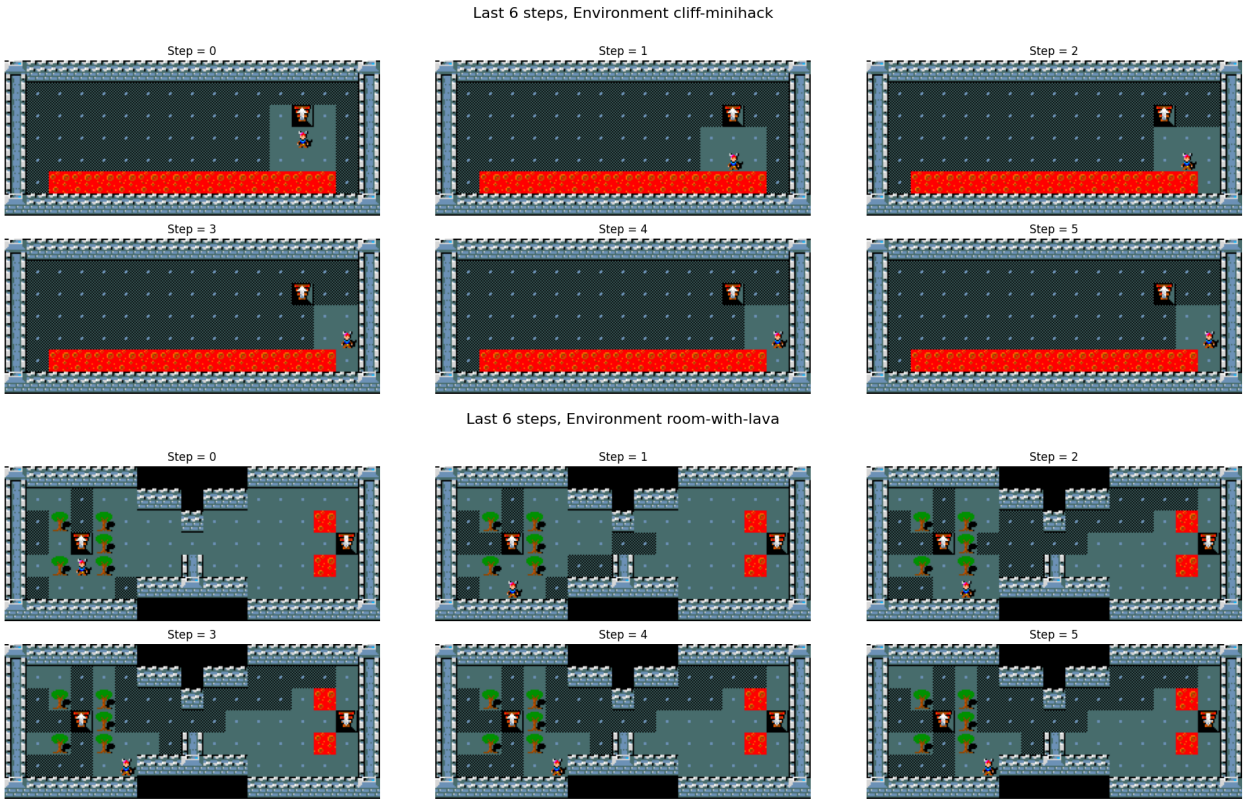


Figure 4: Demonstration of FixedAgent in the room-with-lava and cliff. In this cliff I set Random=True.

Task 2, Reinforcement Learning

In this section, I implement the algorithms:

a) On-policy First Visit Monte Carlo Control

b) SARSA Control (TD on-policy)

c) Tabular Q Learning Control (TD off-policy)

d) Tabular Dyna-Q Control.

Using ε-greedy choice for exploration, incremental updates based on alpha (learning rate) and linear adaptation of ε. I run experiments on the 4 MiniHack environments I showcased above.

Algorithmic Design

For the algorithms I essentially implement the pseudocodes described by the book of Sutton and Barto, with the necessary adaptations (e.g. MC is described for the ε-soft version, I

implement the ε-greedy). The algorithms need implementations for the Q table, the policies after the final improvement (last step of control), as well as the Dyna-Q environment Model.

To implement these I will use Hash tables. The final **policy** is a table with key the state representation and value the corresponding policy (an integer due to discrete policy space). The **Q** table is implemented with having as key the state representation, and value a 1D array containing the estimated Q values for each one of the discrete actions. The **model** has as key a tuple containing both the state and action, and returns a tuple containing the successor state and the reward.

The main thing we should discuss is how we get the agent state representation. The environments return the chars representation as the observation. They can also return the pixel representation, however this needs rendering and greatly increases overhead, and this is why I only use the chars representation.

I begin by cropping the chars representation, so that I can reduce the dimensionality of the corresponding array. This helps, because in order to make the state representation **hashable in an efficient way** I use the **bytes representation**. Having arrays that are smaller in size make the whole hashing procedure more efficient. We should not forget that cropping does not lead to loss of useful information.

Apart from this, I should also note that when *random=True* in the environments, the upwards ladder changes positions in a map in different random settings. The upwards ladder does not offer anything in terms of how the agent interacts with it, it is equivalent to floor tiles. However, the fact that its position changes creates more states that should not exist in the agent's processing procedure and can in fact cause **state space size explosion**. This is a general problem that is more observable later (e.g. with multiple monsters), regardless of the existence of the upward ladder. I **substitute the upwards ladder with a floor** tile to soothe the problem.

Experiments

Before starting with the experiments, I will note some basic stuff about the characteristics of the 4 algorithms I implement and how I expect them to perform.

MC methods need to finish whole episodes in order to do value updates, this can seriously cause problems like make convergence slower, reduce data efficiency etc. The MC methods have higher variance but are unbiased estimators.

TD methods update the values by bootstrapping (recursive equations), SARSA with the actual e-greedy chosen action and Q learning with the optimally (greedy) choice. They are more data efficient, have smaller variance and generally converge sooner. Q learning could be described as less cautious - more bold than SARSA, because it is more biased towards the greedy choices due to the way it bootstraps, and might lead to better (in terms of optimality) policies earlier and with more ease.

Dyna-Q is similar to Q learning but even more data efficient and faster (in terms of convergence for given amount of episodes), because it reuses real experience in simulations for updates, but this can increase CPU overhead. Also, using just a hash table is often not adequate to model more complex environments and this could cause problems too (essentially instead of help the agent).

Empty Room and Room With Monster

We begin running our experiments. In this part, I set the following hyperparameter choices:

$$\text{size} = 5,\ \text{max\_episode\_steps} = 250,\ \varepsilon = 0.1,\ \alpha(\text{learning rate}) = 0.1,\ \text{num\_episodes} = 500,\ \gamma = 1$$

I begin by running the three first algorithms on the Empty Room Environment (Monte Carlo, SARSA, Q learning). The results can be seen on figure 5

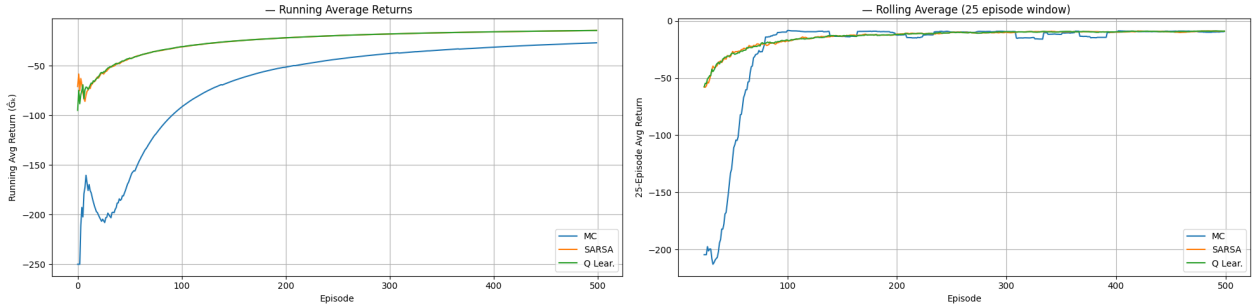


Figure 5: Solving the Empty Room with the 3 algorithms.

I see that they all converge to the optimal policy. However, the MC method has visibly more variance and needs more episodes to reach a good performance, it struggles early on. As we see learning this environment was quite easy, as the task is fairly simple - just go to a fixed location as soon as possible. I will now test the ability of the algorithms to solve a more difficult variant of the square room problem - the case where we also have a monster added (figure 6).

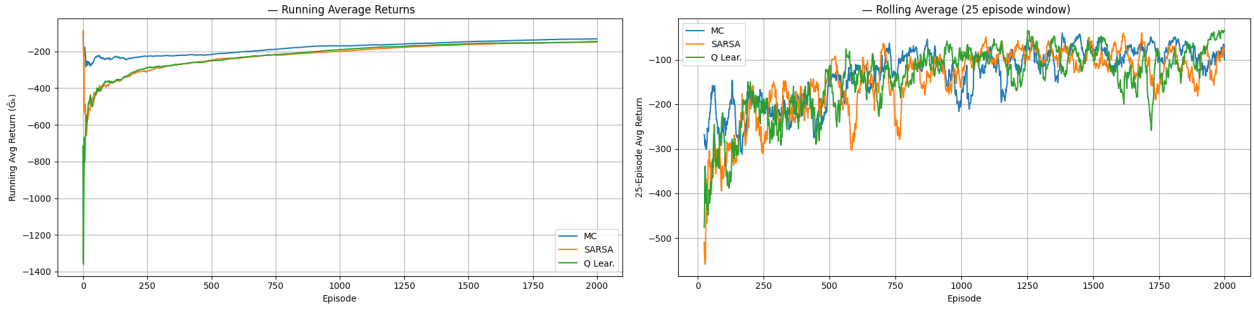


Figure 6: Solving the Room with Monster.

I observe that in the case of the monster the agents seem to have a similar performance from fairly early on, even though now the MC algorithm seems to be slightly better on the first

episodes. As I see this is mainly due to the fact that these maps abort fairly early, at around 100 steps, and it seems like the MC has much more abortions which means that it does not reach the goal. Essentially, the TD methods experiment with the monster and get killed more often early on, while also being able to reach the goal more often.

I also see that the agents still tend to get killed sometimes, even late in the game. I interpret this as being due to the fact that we use a non zero epsilon value, and the agents sometimes explore instead of following the optimal policy and do not avoid the monster properly.

It is also worth noting that the state space is now larger than it was before (625 vs 25). This is due the fact that now the monster can have 25 different locations too, just like the agent. We observe that the addition of the monster makes the state space complexity increase - we should be extra careful with this if we use more monsters, as they can make the state space too large for a simple hashing implementation to handle!

Cliff Environment, Trajectory Depiction

It is now time to see how well our algorithms can handle the cliff environment. The results (for the parameter choices I specify earlier - only eps=0.05 now) can be observed on figure 7.

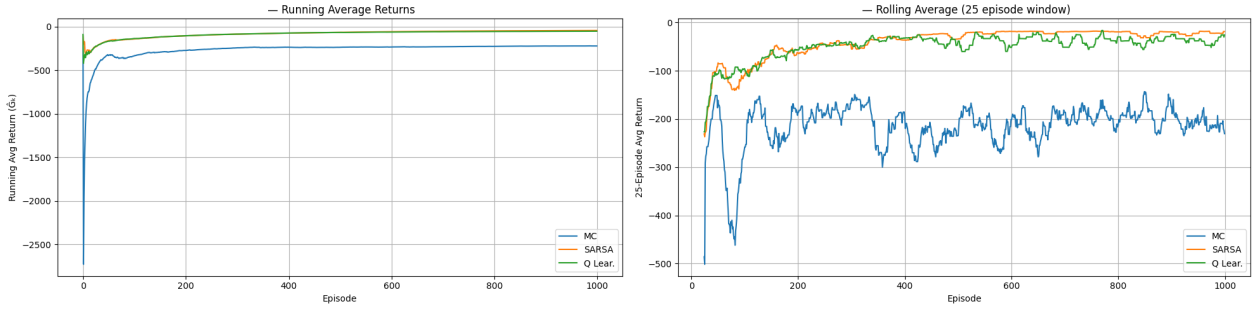


Figure 7: Solving Cliff

I observe that MC has returns around the value of -250, which is equal to the amount of episodes. This means that it only learns how to avoid the lava, and not how to reach the ladder. This is due to the fact that MC actually never reaches the ladder (the probability of reaching it is too small) and it does not get the signal to help it tune the policy properly, a problem greatly exacerbated by the way it updates the values.

On the other hand, I observe that the temporal methods have average returns clearly larger than -100, which means that they have learned both to avoid the lava and reach the goal. However, they do not have exactly the optimal return and I interpret it again to the epsilon value - the forced exploration makes them fall into lava on their way to the ladder.

I also observe that SARSA is slightly better than Q Learning! I think this is a case of when the boldness of Q learning I mention above is a problem, Q learning goes near the lava because it is a shorter path and gets killed from forced exploration more often.

I run the Cliff environment again, this time increasing the max_episode_steps to 500, and increasing epsilon to 0.3 (figure 8).

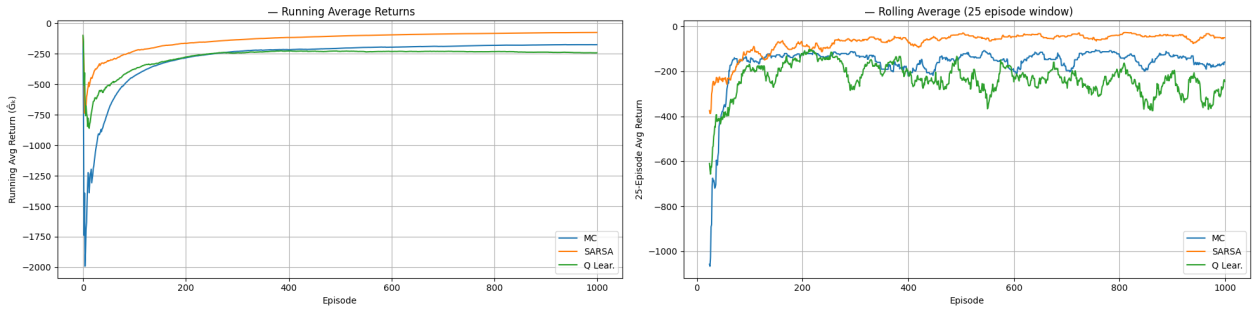


Figure 8: Solving Cliff (500 steps per episode, eps=0.3.)

I observe that the more exploration and the more steps per episode have helped MC actually solve the Cliff map! I also see that Q learning is now even worse than MC, and I say that this is due to the fact that higher epsilon causes the Q learning to actually step to lava even more often than before and destroys its returns!

To more closely observe the policies the 2 TD methods learn, I apply a linear decay of epsilon, with $\varepsilon_{start} = 0.99$ and $\varepsilon_{end} = 0.01$. I show the trajectories that I get in episodes 500 and 999 (the middle and the final one) after following the target policy. The results can be seen on figure 9

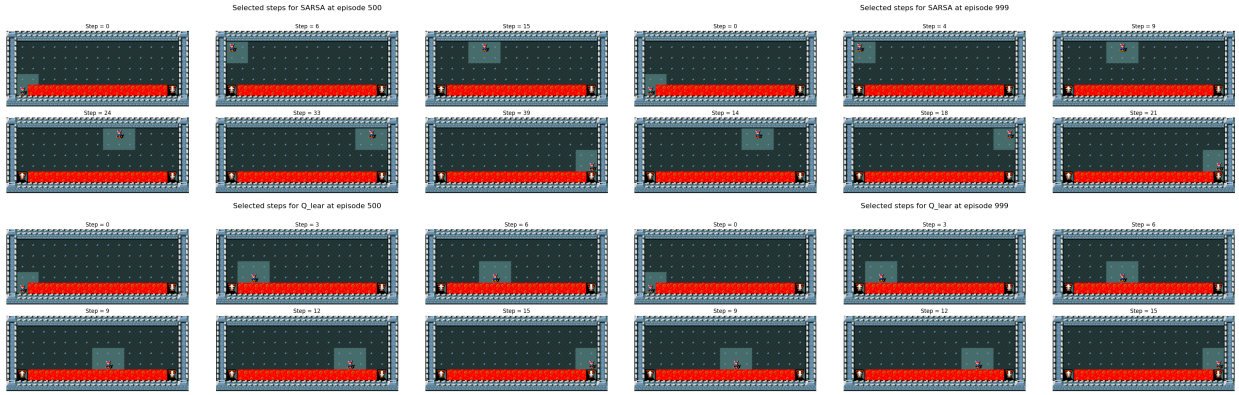


Figure 9: Trajectories shown as 2x3 image figures. SARSA (top) and Q learning (bottom), at episodes 500 (left) and 999 (right).

This figure confirms that Q learning learns the more optimal and bold strategy, to reach the ladder by bordering the lava. Here, where we follow the greedy strategy with no ε the agent does not die and shows peak performance. SARSA on the other hand takes much more cautious strategies to avoid the lava. In the end, where ε is near 0, the agent follows the very cautious strategy of following the walls far apart from the lava.

Room With Lava

The final minihack environment I will work with, using the 4 algorithms I implemented is the Room with Lava. I reset the parameters to the values I specify in the initial experiment. I also run another experiment (second row of 10) in which I increase the learning rate from 0.1 (fairly small) to 0.3 (fairly large) to make the comparison.

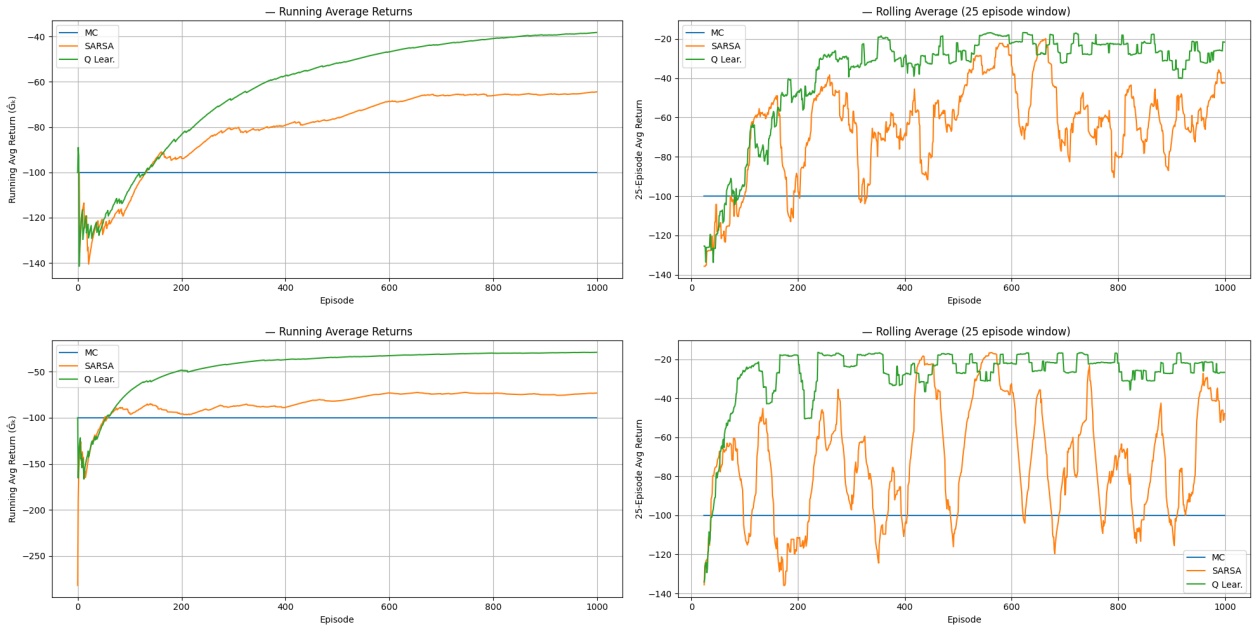


Figure 10: Solving Room with lava. α=(0.1, 0.3) in the top and bottom row respectively

From the first row I observe that the MC again cannot solve the problem, in a fashion similar to the CLIFF environment. The Q learning has returns around -20, I interpret this as it learning the optimal policy but having slighlty reduced returns due to forced exploration ($\varepsilon > 0$). SARSA has smaller returns, which means that it either takes too many steps or falls into lava more often... the extra greediness of Q learning helps in finding a more optimal policy here.

By increasing the learning rate we put more emphasis on the recent experiences and less on the past estimates. This can make training seemingly faster early on (the TD methods surpass the -100 return value earlier with a=0.3) but it can also make the updates and therefore the training more unstable and less smooth, as demonstrated by the higher variance of returns, especially for SARSA! Choosing a fairly smaller learning rate in a relatively more complex environment, like the room with lava seems to be a better choice.

Q Learning vs Dyna-Q

In this part, I run both Q learning and Dyna-Q (figure 11) with the parameters:

size = 5, max_episode_steps = 250, (e_start = 0.99, e_end = 0.01), alpha = 0.1, num_episodes = 1000

gamma = 1, n = 10

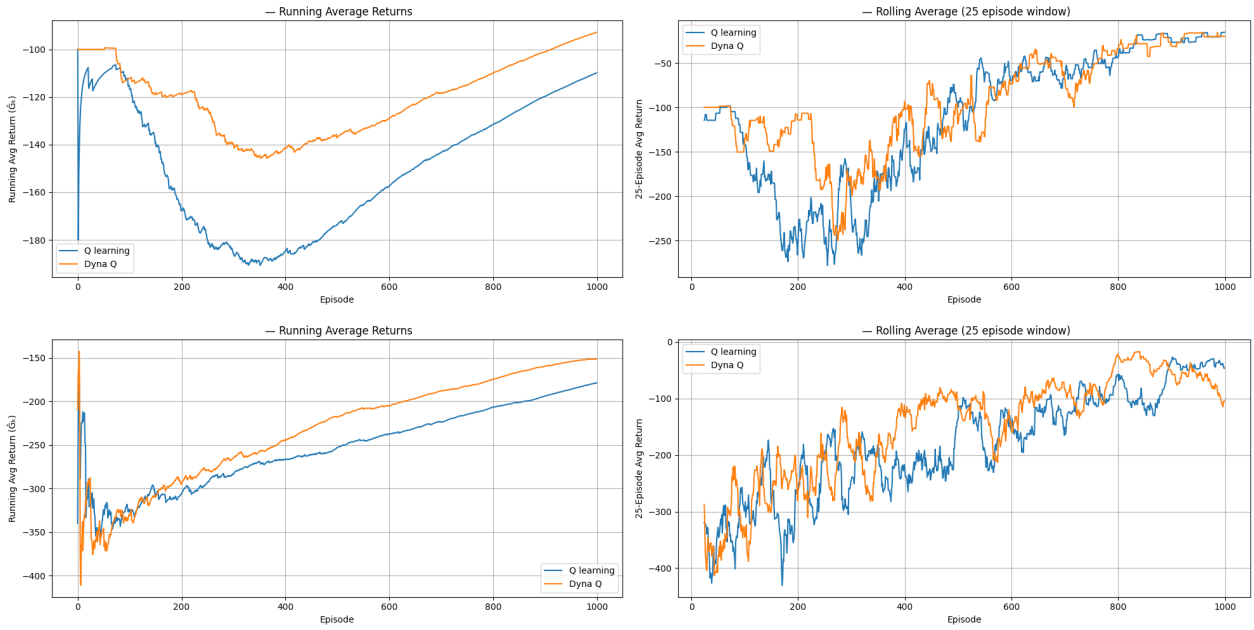


Figure 11: Top row: Room With Lava, t=(23.3, 31.3) for Q learning vs Dyna-Q respectively
Second row: Room With Monster, t=(20.76, 24.79) for Q learning vs Dyna-Q respectively

I observe that Dyna-Q has a better performance than Q learning, especially early on, which is due to the fact that the extra updates from the simulated data help the agent learn more quickly. The differences are less obvious later on, it seems like they find similar policies. We should also note the fact that for the same amount of episodes Dyna-Q is computationally more demanding (check figure caption)!

Deep Reinforcement Learning

It is well known that the tabular implementation of Q learning (the classical method) has a large disadvantage, and this has to do with its lack of generalisation ability. The method essentially only estimates Q values for states it has already seen and it cannot have an abstract point of view and better understand the way the (s,a) pairs and the values are correlated.

Function approximation methods essentially use regression techniques to implement the value approximation. They are done by creating the target values, which actually come by using the same methods as we did in the classical methods. To be specific, in Q learning through function approximation we create our targets through bootstrapping with the max operator, just like in the tabular method. We prefer non linear models, like Neural Networks, because of their

impeccable ability to generalise. By using different architectures we can also more effectively process the observations if given in another format (e.g. CNN to process image observations). We should note however that this generalisation ability is costly in terms of computation and data-efficiency.

With the term "Deep Q-Learning" we refer to techniques that use Deep NNs to implement function approximation with Q-learning. They generally use experience replay (a technique based on randomly sampling minibatches) to mitigate the problem of high correlation between data points (prevalent in RL). Also, due to the fact that targets are changing and not constant (Recursion due to bootstrap), we use another network that is updated more slowly than the Q network to help us have a resemblance of stationarity in the targets (called the target network).

In double DQL, we implement the target network in a way such that it implements Q learning, thereby reducing the problem of the Q-value overestimation bias. Dueling Deep Q Networks split the Q estimation function in two parts, the value stream (which shows the state value) and the advantage stream (which shows how much we benefit from using a specific action), which can speed up learning and improve performance.

In my implementation, I use stable-baselines3 and implement a Deep Q network with Double and Dueling implementations. I use an MLP of [256,128] for the network. I pick most hyperparameters equal to the default ones.

Apart from Deep Q learning, we can also use NNs to implicitly approximate the optimal policy itself, with methods called policy gradient. Actor-Critic methods combine the function approximation network (actor) with another network which estimates the state values (critic). This can help us implement training with baselines (i.e. implementing action advantages) but also updates through bootstraps.

Actor-Critic algorithms have the disadvantage of being sensitive to the weight updates, i.e. fairly small updates can cause large changes in how the model operates. To mitigate this problem, we can use techniques like Proximal Policy Optimisation, that use a clipped surrogate objective and implement an idea similar to trust regions in a manner equivalent to soft optimisation (vs hard that exists in Trust Regions) - PPO implement this idea without being too computationally demanding. Apart from this, we also generally try to encourage stochasticity in the policy in order to incentivise exploration, this is often done by adding entropy regularisation (i.e. adding the policy entropy in the optimisation in order to encourage higher randomness).

I use stable-baselines3 and implement a PPO actor-critic method with entropy regularisation, with an architecture similar to the one I had on the DQN. Most hyperparameters are either default or similar to the ones I had in DQN for a better comparison. The results from running the algorithms can be seen on figure 12.

I run the Deep Models on the GPU of the interactive node of the Flemish Supercomputer Center (VSC).

In the Empty room, the times needed are 7, 66 and 37 seconds for Tabular, DQN and PPO

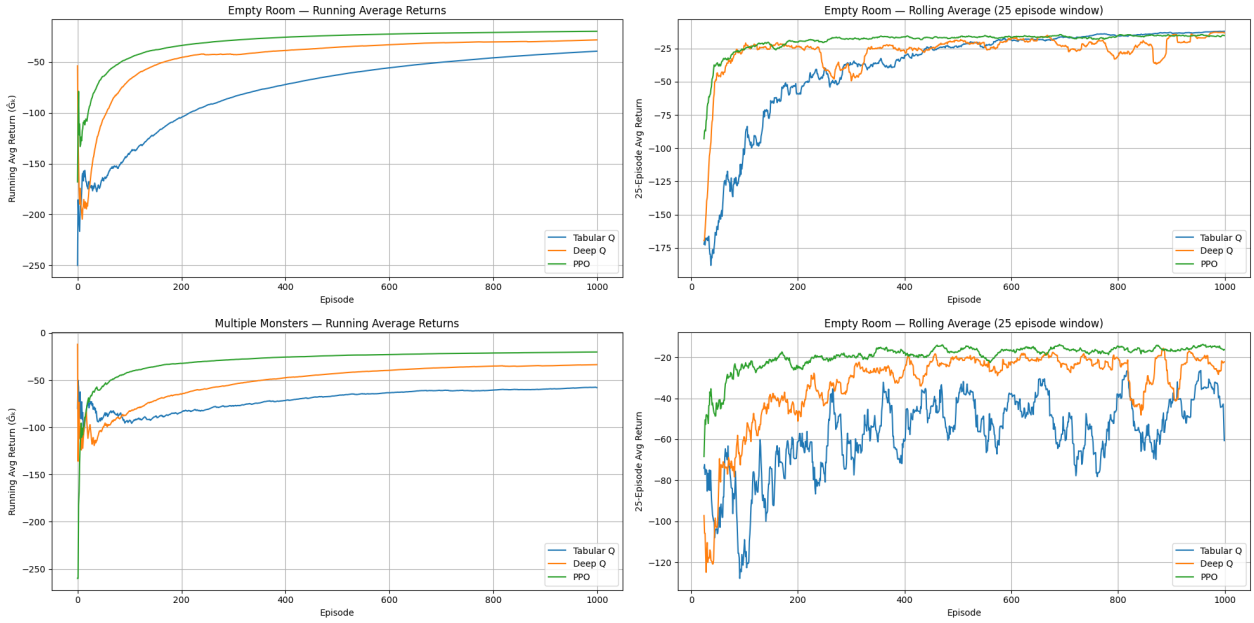


Figure 12: Running the Tabular Q, the DQN and the PPO algorithms for Empty Room (7x7) and Room With Multiple Monsters

respectively. In the Room with multiple monsters, the times are 11, 170 and 110 seconds respectively. The tabular method is clearly faster than the Deep Learning methods, while the PPO is faster than the DQN, at least with my hyperparameter choices.

In the first map, I see that the tabular method needs more episodes to converge to the optimal policy. However, from one point onwards it is more stable than the DQN! I interpret this as being due to the fact that the state space is small and the hash table is adequate to properly and reliably estimate the q values - the DQN has more capacity but is a more unstable mechanism (around 40k weights, much larger amount of parameters to tune). PPO is the fastest and most stable of them all.

In the second map however we can clearly see the weakness of the tabular method, which simply cannot reach the solution quality we have with the DL methods. This is not surprising, as the state space is now larger (more than 16000 possible states), and having generalisation abilities might come in helpful. PPO is the most stable and fast of them all, and reliably receives returns around -16, which is quite near the optimum for the 7x7 room without getting killed. Perhaps in this room, where we have the less predictable element due to the monsters, being able to implicitly model stochastic strategies really helps.